# Documentation

Amin Haidar, Samuel Pearce

## 1. Table of contents



## 2. Project Description

For our project we originally had a text-based strategy game, to be run in the console in mind. This proved to be too exciting of a prospect, and we were unable to restrain ourselves from adding too many cool features. This idea was shelved for another time.

Our design pivoted to a simpler text-based game like Zork[1]. We discussed what the story should be and settled on a game where you have to escape from TBZ. Each level is a floor and each floor gets you one step closer to the exit on the ground floor. Then some doors will be locked and you must look around the place to find keys and other items as well as talk to NPCs (Non Player Characters). Our next thought was to give the player a "hunger" and "thirst" meter. If either one reaches zero, the player dies and must start the level again. The meters are replenished by eating and drinking assorted food the player finds around TBZ, but one shouldn't be too quick to drink all that water. If your "urine" meter fills to

[1] http://textadventures.co.uk/games/play/5zyoqrsugeopel3ffhz_vq

100%, then you will wet yourself and none of the NPCs will want to talk to you and you could be blocked from progressing. To reduce the "urine" meter, the player must find a bathroom in which to relieve themselves.

## 3. Milestones

### *3.1. Project Planning*

The first milestone will be reached upon having created all UML diagrams and the test-cases. This milestone marks the beginning of practical progress. Before this point no code will have been written yet. Although, after this milestone, some elements of the diagrams may change in accordance with the projects progress.

### *3.2. Classes Created and Tested*

This milestone will be reached once all classes have been written, have their full functionality and have been unit-tested. This milestone means the bulk of the project is complete. Only the integration testing and bug-fixing is left of practical work on the project.

### *3.3. Testing Complete*

This milestone will be reached when all the integration tests are completed. This means that the application itself is complete and fully functional. At this point the only remaining work is finalising the documentation.

### *3.4. Project Complete*

This milestone will be reached after all previous milestones have been reached and the project`s documentation is complete and accurate. The only work after this milestone is popping the champagne.

## 4. Dependencies

### *4.1. SimpleTUI - v2.0.0*

GitHub Page[2]

[2] https://github.com/Sam36502/SimpleTUI

This library wouldn’t have been necessary, but it saves us a lot of time on boring and boilerplate input validation. Not to mention: The library itself was made by Sam previously. Therefore, no code was actually taken from any third-parties for the input validation.

## 5. UML Class Diagram

> Insert completed diagram here

## 6. UML Sequence Diagram

> Insert completed diagram here

## 7. UML Use Case Diagram

> Insert completed diagram here

## 8. Use Cases

---

#### UC01 - Enter Command

**Actor**

Player

**Basic Flow**

The player enters text into the console and presses the carriage return key. If the command matches a command in the system, then the command is executed. If the command doesn’t match any known command, then the player is informed, that this command wasn’t recognised.

If the action made the player perform something, i.e `move`, `use` or `pick-up`, then two points are taken off their "thirst" meter and one point is taken from their "hunger" meter.

**Preconditions**

The game has been started and the player has made it past the main-menu.

**Termination**

The player has performed an action in the game or are informed, why this could not be performed.

---

### UC02 - Move Player

**Actor**

Player

**Basic Flow**

The player enters the `move` or `mv` command into the console with a single argument that is either `n`, `north`, `w`, `west`, `e`, `east`, `s` or `south`. If the syntax is correct and there exists an open path between the current room and the room in the direction provided, then the player will be moved to the room in the given direction. If such a movement is not possible, then the user is informed that they are unable to move in that direction.

**Preconditions**

The player must be in a level and in a room with an open path to an adjacent room.

**Termination**

The player is now in the room in the direction that they selected from the first room.

---

### UC03 - Pick up Item

**Actor**

Player

**Basic Flow**

The player enters the `pick-up` or `pu` command into the console with a single argument. This argument must be the name of an item in the room.

If the named item exists in the room, then it is removed from the room and put into the player`s inventory. If the Item cannot be found in the current room, the player is informed, that no such item exists.

**Preconditions**

The player must be in a room that has at least one item in it.

**Termination**

The item is no longer present in the room and has been added to the user`s inventory.

---

### UC04 - Consume Item

**Actor**

Player

**Basic Flow**

The player enters the `eat`, `drink` or `cs` command into the console with a single argument. This argument must be the name of a consumable item in the player`s inventory. If the player has this item and the item is consumable, then the item is consumed and the user`s "thirst", "hunger" and "urine" meters are updated accordingly. If the user does not posess the named item, then they are informed thusly. If the item they do name is owned, but not consumable, then they are told, they consume the given item.

**Preconditions**

The user has a consumable item in their inventory.

**Termination**

The item is removed from the user’s inventory and their necessity meters are updated accordingly.

---

### UC05 - Interact with Interactable

**Actor**

Player

**Basic Flow**

The player enters the `use`, `interact`, `uo` or the `in` command into the console with two arguments. The format of the command is:

`use <item> on/with <interactable object`

If the player has the named item in their inventory and the interactable object exists inside the current room, then the interaction takes place. Otherwise, the player is simply informed that they are unable to perform this interaction.

**Preconditions**

The player is in a room with at least one interactable object and have at least one item in their inventory.

**Termination**

The item is no longer in the user`s inventory and the interaction takes place.

---

### UC06 - Get Help

**Actor**

Player

**Basic Flow**

The user enters the `help` or `?` command into the console. Upon executing this command, the console will display a list of all currently available commands as well as a list of their necessary arguments. If the user enters the command with an argument, a more in-depth explaination is outputted along with some examples of how to use that specific command.

**Preconditions**

The user must be able to enter commands and be at least past the main menu.

**Termination**

The user is given the information they requested.

# 9. Unit Test Cases

## *9.1. Item*

**Table 1. Item Test Cases**

| Input | Expected | Output | Approval |
|---|---|---|---|
| Item("") | ItemNameExc. | ItemNameExc. | OK |
| Item(" ") | ItemNameExc. | ItemNameExc. | OK |
| Item("123") | ItemNameExc. | ItemNameExc. | OK |
| Item("abc") | - | - | OK |
| Item("", 1, 1) | ItemNameExc. | ItemNameExc. | OK |
| Item("abc", -1, -1) | ItemReplExc. | ItemReplExc. | OK |
| Item("abc", 101, 101) | ItemReplExc. | ItemReplExc. | OK |
| Item("abc", 0, 100) | - | - | OK |

## *9.2. Door*

**Table 2. Door Test Cases**

| Input | Expected | Output | Approval |
|---|---|---|---|
| Door("a", Item("key")) | - | - | OK |
| canUseWith(Item("afa", 1, 1)) | false | false | OK |
| canUseWith(Item("ath")) | true | true | OK |
| isLocked() | true | true | OK |
| useWith(Item("abc")) | "That won’t work…" | "That won’t work…" | OK |
| useWith(Item("key")) | "The key clicks…" | "The key clicks…" | OK |
| isLocked() | false | false | OK |

## 9.3. Room

**Table 3. Room Test Cases**

| Input | Expected | Output | Approval |
|---|---|---|---|
| Room("rm1", "A room.") | - | - | OK |
| Room("rm2", "Another room") | - | - | OK |
| getOpposite(WEST) | EAST | EAST | OK |
| parseDirection("NoRtH") | NORTH | NORTH | OK |
| parseDirection("w") | WEST | WEST | OK |
| rm1.setNext(NORTH, rm2) | - | - | OK |
| rm1.getNext(NORTH) | rm2 | rm2 | OK |
| rm1.canGo(NORTH) | true | true | OK |
| rm2.canGo(SOUTH) | false | false | OK |
| rm1.setDoor(NORTH, Door("d", Item("key"))) | - | - | OK |
| rm1.canGo(NORTH) | false | false | OK |

## 9.4. FileHandler

*(Shortened to FH for convenience)*

For the purposes of this test, it is assumed, that there is a file named "test.txt" in an adjacent folder to the executable. The file "nonexistent.txt" does not exist.

The contents of "test.txt"

```
#This is a comment and will be ignored
This is invalid and will be ignored
prop1=Hello! #End-of-line comment
prop2 = 42
```

**Table 4. FileHandler Test Cases**

| Input | Expected | Output | Approval |
|---|---|---|---|
| fherr=FH("nonexistent.txt") | - | - | OK |
| fh=FH("test.txt", "test", WARNING) | - | - | OK |
| fh.getFilename() | test.txt | test.txt | OK |
| fh.loadFile() | - | - | OK |
| fh.getProp("prop1") | Hello! | Hello! | OK |
| fh.getProp("prop2") | 42 | 42 | OK |
| fh.hasProp("prop1") | true | true | OK |
| fh.getPropInt("prop1") | [WARNING]… | [WARNING]… | OK |
| fh.getPropInt("prop2") | 42 | 42 | OK |
| fherr.getProp("prop1") | InvalidPropExc. | InvalidPropExc. | OK |
| fherr.hasProp("prop1") | false | false | OK |
| fherr.loadFile() | (Sys Exit) | (Sys Exit) | OK |

## *9.5. Level*

This unit test will mainly return empty values. To test how it works in conjunction with the other classes, see the "Level Loading" Integration test

**Table 5. Level Test Cases**

| Input | Expected | Output | Approval |
|---|---|---|---|
| Level(0) | LevelNumberExc. | LevelNumberExc. | OK |
| Level(26) | LevelNumberExc. | LevelNumberExc. | OK |
| Level(1) | - | - | OK |
| lvl.findRoom("nonexistent") | RoomNameExc. | RoomNameExc. | OK |
| lvl.loadMap("nonexistent") | (Sys Exit) | (Sys Exit) | OK |
| lvl.getNumber() | 1 | 1 | OK |
| lvl.getRooms() | (Empty Array) | (Empty Array) | OK |

# 10. Integration Test Cases

# 11. Work Journal

## *11.1. Day - 1*

1/13/2019

---

We began by Setting up our documentation method. In our case, we decide to use Asciidoc, as we'd rather not spend our time styling the documentation, when it can all be automated with some basic tools. Not to mention the single simple text format can be exportet to many different document types such as PDF, HTML and DocBook. This means we would also have the option of hosting our documentation as a website.

Next we began making the UML diagrams. We both discussed the class structure and what we would need to store the level structure and item information. Once again, we had many good ideas to add, but reeled in our ambitions for the first version and decided to add the new features later in a version 2.0 if we got the time.

Once we had the class diagram, we split up to work on the use-case and sequence diagrams. Amin began writing the use-cases and created a diagram, that we both agreed, protrayed our project faithfully. (Although both our experience with UML is rather limited)

Sam then did some research on sequence diagrams and got to work creating them. At first we weren't quite sure how to visualise conditionality in the diagram, but we eventually got the hang of using alternate paths.

Finally, we were finished with most of the planning stage and planned to start programming next time.

## *11.2. Day - 2*

13/10/2019

---

Over the weekend we both chose some Classes to create and individually unit-test. We occasionally had some issues and had to discuss how the files should be loaded. However, we were able to standardise our system so that if we wanted to add any more new features, then it is easily expandable.

Amin took the Item, Interactable, Door and Room classes and implemented and tested them. Sam took on the Level and FileHandler classes. These were not as simple as the others. The level class contained one rather monolithic function for loading all the world attributes into the rooms from a file.

Due to the Item class being created first it was documented far more in-depth than the other ones. it was documented the most in-depth. We discussed writing a seperate documention for each class in order to more completely document each part of the game.

Sadly, the time is lacking, due to our early slacking in the project and we will keep the shortened documentation. We will always have time to add the improved documentation later.

We hope to finish off the remaining classes during our free time over the next week.